XQuery Streaming *à la Carte*

Mary Fernández
AT&T Research
mff@research.att.com

Philippe Michiels
Universitat Antwerpen
philippe.michiels@ua.ac.be

Jérôme Siméon
IBM Research
simeon@us.ibm.com

Michael Stark
TU Darmstadt
stark@mis.tu-darmstadt.de

Abstract

Existing work on XML query evaluation has either focused on algebraic optimization techniques suitable for XML databases, or on algorithms to efficiently process XML messages represented as a stream of parsing events. In practice, complex applications often must handle both. In this paper, we develop a physical algebra that combines streaming operators with other standard relational and XML operators. Our physical model includes marked XML streams, which permit efficient XPath evaluation, but can only be consumed once. This constraint restricts the use of streaming operators to fragments of a query plan that only access data using depth-first traversal. We develop static analysis techniques to decide which fragment of a plan can be streamed. Our experiments demonstrate the benefits of blending streaming with other evaluation techniques.

1 Introduction

XQuery 1.0 [5] is increasingly used in diverse application environments from querying persistent XML repositories, to processing XML messages in Web-service applications, to integrating data from multiple XML sources. Complex applications often have to query both long-lived and transient XML data sources. Existing XML query evaluation techniques, however, draw from two fixed menus: algebraic optimization techniques [4, 23, 25, 26] with algorithms for persistent data, or streaming[1] algorithms [3, 9, 13, 19, 20] for transient data. In this paper, we develop a physical algebra for XQuery in which streaming algorithms and traditional optimization techniques can be used *à la carte* based on application needs.

The bidding Web service in Figure 1 illustrates the characteristics of queries in applications that must handle diverse XML data sources. In this example, there are two data sources: A persistent data store in `pdb.xml` contains the profiles of auction bidders, and a Web data source, denoted by the variable `$bids`, continuously produces bid

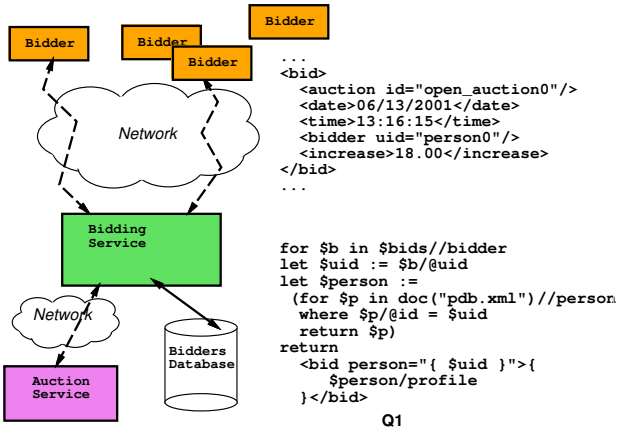

Figure 1. Auction Web Service

events. For each input bid event, the query Q1 produces a new bid event that includes the bidder's profile and that is sent for further processing to the main auction service.

In essence, query Q1 expresses a join between a streaming source and a local repository. Figure 2 depicts the physical plan we would like to obtain. The fragments of the plan that operate over XML-token streams are in gray boxes, whereas the other fragments operate over materialized XML. On the fragments' boundaries, the Load and Export operators convert between those two representations. When accessing the input bid events, the plan uses a streaming path-expression algorithm that avoids materialization of the bid events and that can initiate query evaluation as soon as the bid events arrive. When accessing the bidders' profiles in the persistent store, the plan uses a tree pattern algorithm [8, 14, 18, 24], that efficiently evaluates path expressions on indexed documents. To handle the nested query, query decorrelation techniques [23, 26] are used to produce a join between the two sources. Because the query result is sent to another Web service, it should be generated as an XML-token stream, avoiding unnecessary creation of XML trees in memory or on disk. The physical algebra and compilation techniques proposed in this paper produce a query plan that achieves these goals.

Most of the work on XML streaming focuses on algo-

[1] Streaming means here query-evaluation techniques that are applied to sequences of XML tokens, similar to those produced by a SAX parser.

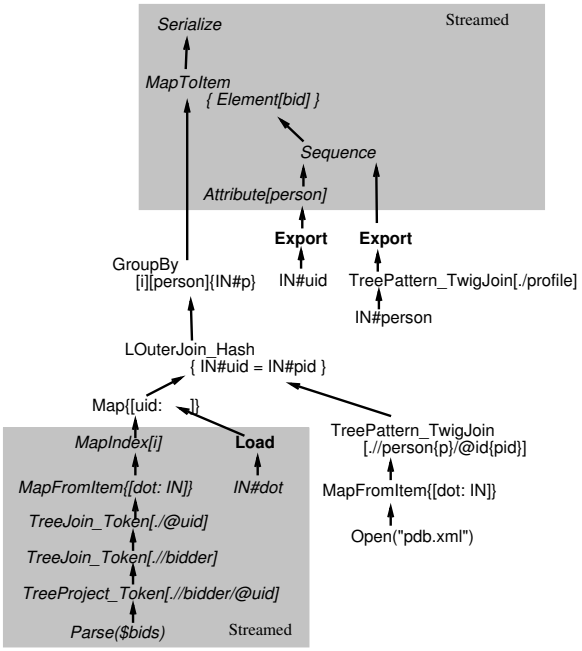

Figure 2. Physical query plan for Q1

rithms for restricted languages [3, 19, 20], which are difficult to use in an algebraic compiler. More recent work on the XQRL [13], SPEX [9] and XStream [16] systems support streaming techniques for larger fragments of XQuery, but use an evaluation model that is difficult to integrate in a traditional algebraic compiler. Fegaras [6, 12] and Rudensteiner's [27] approaches incorporate stream operators in an algebra. The latter is restricted to a navigational subset of XQuery and uses buffering pushdown automaton on streams. The originality of our approach is that it combines streaming algorithms with other traditional evaluation techniques, like index-based access, and join and query-unnesting optimizations. Our approach relies on a physical data model that includes tuples and materialized-tree and streamed representations of XML, and on a physical algebra that combines operators on both XML representations and supports conversion between them. When referring to the streamed representation, we use the terms XML-token stream and XML-token cursors interchangeably.

Our physical algebra is designed to meet the following requirements. First, it can serve as the physical layer for an existing logical algebra [26]. For that reason, it requires few changes to early phases of compilation and should be applicable in other XQuery algebraic compilers [10, 17, 23]. We refer the reader to the substantial work on logical compilation [10, 17, 21, 23, 24, 26]. Here, we focus on the definition of a suitable physical data model and physical algebra for an existing algebra [24, 26].

A second requirement is for streaming operators to support pipelined evaluation, following the evaluation model of modern relational engines. We satisfy this requirement by defining a small set of *micro-operators* on mutable cursors, the majority of which are non-buffering. Non-buffering operators can be evaluated in constant space and time linear in the size of their input. We define our streaming algebra in terms of these micro-operators, for which correctness and complexity properties are easily defined. These operators, however, destructively modify their cursor inputs, which means cursors can be consumed in a plan at most *once*. The mutability of cursors supports efficient access to data, but complicates meeting our last requirement.

Our last requirement is that every physical plan produced by code selection is correct, that is, it implements the logical semantics of the corresponding logical plan. In Section 4, we give a formal definition of plan correctness and present whole-plan analyses that identify the fragment of a plan that may be safely implemented by streaming operators. In particular, these analyses guarantee that every XML-token cursor used in a plan is consumed exactly once.

We implemented our physical algebra and static analyses in the Galax XQuery engine and applied a simple code-selection heuristic to "stream as long as possible". In Section 5, we present an experimental evaluation of the streaming fragment of our physical algebra and show that streaming is effective on a variety of benchmark queries. In particular, we present micro-benchmarks that verify the linear scalability of streaming operators in both query and data size, and we measure the impact of streaming on the XMark benchmark suite and on our motivating query Q1.

2 Preliminaries

We begin by defining a model for cursors, then define our physical data model that supports materialized and streamed XML. A *cursor* (*list*) is a mutable (immutable), ordered sequence of homogeneous values. A cursor is destructive in that accessing the next token also removes it from the cursor. This property means that cursors can be used for both relational operators and XML streaming operators and that they only use constant space. As a result, they may be consumed only once in sequential order.

$\mathbb{C}(\alpha)$ denotes a cursor containing values of type α. Table 1 gives the signatures for micro operators on $\mathbb{C}(\alpha)$. The fromList operator takes a list of values and constructs a cursor over the same values. The next operator is destructive: It takes a cursor, removes the first value from the cursor, and returns it. next is undefined on the empty cursor. The peek operator is non-destructive: It takes a cursor and returns the current value from the cursor. peek returns *empty* on the empty cursor. The other operators in Table 1 are defined later in the paper.

2.1 Physical Data Model

Our physical data model is based on the logical data model described in [26] and is given in Table 2. A physical value is either a physical XML value or a physical

Table 1. Cursor and token-cursor operators

Section 2	fromList	: $\mathtt{L}(\alpha) \rightarrow \mathtt{C}(\alpha)$
	next	: $\mathtt{C}(\alpha) \rightarrow \alpha$
	peek	: $\mathtt{C}(\alpha) \rightarrow \alpha \mid empty$
	load	: $\mathtt{C}(Tok) \rightarrow \mathtt{L}(Tree)$
	export	: $\mathtt{L}(Tree) \rightarrow \mathtt{C}(Tok)$
	unfold	: $\mathtt{C}(Tok) \rightarrow \mathtt{C}(Tok)$
	parse	: $() \rightarrow \mathtt{C}(Tok)$
	serialize	: $\mathtt{C}(Tok) \rightarrow ()$
Section 3.1	concat	: $\mathtt{C}_1(\alpha) \times \mathtt{C}_2(\alpha) \rightarrow \mathtt{C}(\alpha)$
	compose	: $\mathtt{C}(Tok) \times \mathtt{C}(Tok) \rightarrow \mathtt{C}(Tok)$
	unmark	: $\mathtt{C}(Tok) \rightarrow \mathtt{C}(Tok)$
Section 3.2	$\mathrm{nav}_{[step]}$	: $\mathtt{C}(Tok) \rightarrow \mathtt{C}(Tok)$
	markmap	: $(\mathtt{C}(Tok) \rightarrow \mathtt{C}(Tok)) \times \mathtt{C}(Tok) \rightarrow \mathtt{C}(Tok)$
	prune	: $\mathtt{C}(Tok) \rightarrow \mathtt{C}(Tok)$
Section 3.3	$\mathrm{map}_{[x]}$	: $(\alpha \rightarrow \beta) \times \mathtt{C}(\alpha) \rightarrow \mathtt{C}(\beta)$
	split	: $\mathtt{C}(Tok) \rightarrow \mathtt{C}(\mathtt{C}(Tok))$

Table 2. Physical data model

Value	::=	*Xml* | *Table*
Xml	::=	$\mathtt{C}(Tok)$ | $\mathtt{L}(Tree)$
Table	::=	$\mathtt{C}(\tau)$ | $\mathtt{L}(\tau)$
τ	::=	[$\mathtt{q}_1$: *Xml*, ..., $\mathtt{q}_n$: *Xml*]
Tok	::=	$startElem(\mathtt{q}, \mathtt{M}?)$ | $endElem$
	|	$text(String)$ | $atomic(a)$ | $hole$

table. A physical XML value, *Xml*, is either a cursor of XML tokens, $\mathtt{C}(Tok)$, or a list of tree values, $\mathtt{L}(Tree)$, which provide the abstract interface defined on nodes and atomic values in the XQuery Data Model. A physical table, *Table*, is either a cursor of tuples, $\mathtt{C}(\tau)$, or a list of tuples, $\mathtt{L}(\tau)$. A physical tuple, τ, is a record with fields containing physical XML values. Physical tuples are denoted by $[\mathtt{q}_1 : Xml; \ldots; \mathtt{q}_n : Xml]$, where $\mathtt{q}_i$ are field names.

An XML token, Tok, is a parsing event like that produced by a SAX parser. A $startElem$ token denotes the beginning on an element and includes the element's qualified name $\mathtt{q}$ and an optional mark $\mathtt{M}$. An $endElem$ denotes the end of an element. A $text$ token denotes a text node and takes a string value, and the $atomic$ token takes an atomic value.[2] Lastly, the $hole$ token denotes a location in a token cursor at which another token cursor may be injected.

2.2 Marked Token Cursors

Token marks and holes support efficient implementation of navigation and constructor operators, respectively. In particular, a sequence of tokens delimited by a *marked* $startElem$ and its corresponding $endElem$ denotes an *output* tree, which may be the result of parsing or an axis step. Marked $startElem$ tokens may be nested as a result of axis steps that yield multiple nodes in an ancestor-descendant relationship. For example, the token cursor in the middle of Figure 3 contains nested, marked tokens, possibly resulting from a descendant-or-self step like `//section`. The tokens and their corresponding end

[2]For space reasons, we omit attributes, comments, and processing instructions from the presentation.

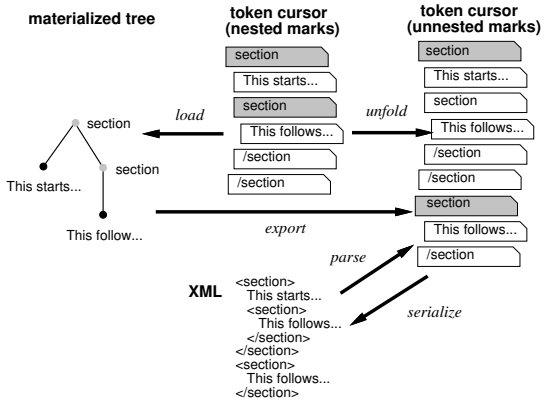

Figure 3. Physical representations

tokens delimit two output trees. In our framework, the boolean mark is sufficient, but if we were to employ more sophisticated streaming operators, marks could be generalized to integer stream levels [12].

2.3 Representation Conversion

Next, we describe the micro-operators that convert between token cursors with and without nested marks, and later, give the actual algorithms for navigation over marked token cursors. We focus on load and unfold, which are the only micro-operators whose space complexity is not constant and whose time complexity may not be linear.

Figure 3 depicts the load, export, unfold, parse, and serialize micro-operators. Marked tokens are highlighted. The load operator takes a marked token cursor and yields a materialized tree. In Figure 3, the nested `section` elements are marked. After materialization, the highlighted `section` elements, corresponding to marked `section` tokens, are returned by load. The load materializes a tree bottom-up, therefore it blocks pipelining in a plan. Moreover, its space and time complexity are both linear in the number of input tokens.

The unfold operator takes a marked token cursor, possibly with nested marks, and yields a token cursor in which all marks are at the top-level by copying marked subsequences. If marked tokens are not nested, unfold simply copies its input to its output. When a nested mark is first observed, the token sequence delimited by the marked token and the corresponding $endElem$ is copied into a buffer, and the buffer offsets of all marked $startElem$s within the copied sub-sequence are recorded. For example, in Figure 3, the marked token sequence for `<section>This follows...</section>` is buffered during unfolding. Once the top-most token sequence containing one or more marked subsequences has been emitted, all the marked sub-sequences are emitted in document order. In our example, the buffered token sequence `<section>This follows...</section>` is emitted after the marked token sequence that contains it is emitted.

Although unfold may be pipelined, its worst-case time complexity is quadratic in the size of its input and occurs when every $startElem$ is marked. In particular, if the input is a tree with n nodes and maximal height $h = n$, then unfold produces n streams of length n.

The export operator takes a list of tree nodes and in a depth-first, pre-order traversal of the materialized nodes, generates a token cursor. The serialize and parse operators convert unfolded token cursors to/from XML. These and the remaining micro-operators all have constant memory complexity and can be fully pipelined.

3 Physical Algebra

In this section, we present a physical algebra for the logical algebra proposed in [26]. Since all those operators have a standard or straightforward implementation over materialized XML, we focus here on the definitions of operators that produce and consume token cursors.

A physical algebraic operator is written:

$$\mathsf{POp}[\mathsf{s}_1, \ldots, \mathsf{s}_i]\{\mathsf{POp}_1, \ldots, \mathsf{POp}_i\}(\mathsf{POp}_1, \ldots, \mathsf{POp}_k)$$

where POp is the operator name; s_i's are static parameters of the operator; POp_i's are dependent sub-operators; and the POp_k's are input (or independent) operators. A sub-operator is *dependent* (*independent*) with respect to a given operator POp, if its evaluation does (does not) depend on the evaluation of other sub-operators of POp. For dependent operators, IN denotes the input XML value or tuple. For example, $\mathsf{MapFromItem}_{[i]}\{\texttt{[x:}i\texttt{]}\}\texttt{((0,1))}$ yields the table `([x:0],[x:1])`. The tuple-constructor operator `[x:`i`]` is dependent since its evaluation depends on the independent input of MapFromItem.

Table 3 lists all the physical operators over streams, giving their signatures and definitions. Many operators are *polymorphic* in their input types. For example, Select takes tuples containing XML-token cursor or tree representations of XML values. The signatures and implementations for those operators are based on standard relational algorithms, except for MapFromItem.

An operator's signature includes the physical types of its sub-operators and of its output. For example, Parse takes a URI and returns an XML-token cursor that results from parsing the document denoted by the given URI. Load takes an XML-token cursor and returns the corresponding physical tree representation. Parse, Serialize, Load, and Export are defined in terms of the micro-operators described in Section 2. Type operators Validate and TypeMatches are included in Table 3 for completeness. Our system implements these operators on type-annotated token streams, similar to those proposed in [13]. Limited space prevents us from giving the corresponding algorithms. In the rest of this section, we give detailed definitions for the constructor and navigation operators.

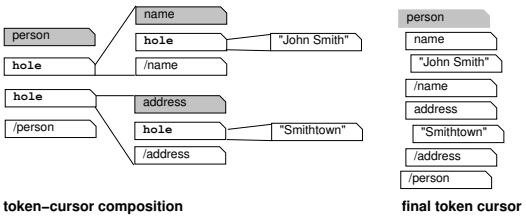

Figure 4. Token-cursor composition

3.1 Constructors

The Sequence operator is defined simply in terms of the concat micro-operator, which consumes two cursors of the same type and returns a new cursor containing the values in the first cursor followed by the values in the second.

The Element operator is more interesting. It is defined in terms of the compose operator, which takes one token cursor with i holes ($i \geq 1$) and a second token cursor with j holes. It yields all the tokens in its first argument up to the first hole, at which point it yields all tokens in its second argument, "filling" in the hole. After consuming its second argument, compose yields the remaining tokens in its first argument, producing a token cursor with $i + j - 1$ holes. This definition permits constructors to be fully pipelined in a plan containing other token-cursor operators. To illustrate, the nested constructor expression:

```
<person>
 <name>{"John Smith"}</name>
 <address>{"Smithtown"}</address>
</person>
```

is implemented by the following plan of micro-operators:

```
compose(
 fromList([startElem(person,M), hole, endElem]),
  unmark(unfold(concat(
   compose(fromList([startElem(name,M),
        hole, endElem]),
        unmark(unfold(fromList([text("John Smith")])))),
   compose(fromList([startElem(address,M),
        hole, endElem]),
        unmark(unfold(fromList([text("Smithtown")])))))))))
```

Figure 4 depicts the above composition. Note that newly constructed elements are themselves marked, but that their content is unfolded and unmarked. Unfolding enforces the logical constraint that a newly constructed element copies its argument. Marking the newly constructed element permits the resulting token cursor to be pipelined into other operators, e.g., navigation.

3.2 Navigation operators

Our physical algebra has two navigation operators defined on marked-token cursors: TreeProject and TreeJoin. Logically, tree projection takes a tree and a set of path expressions and returns a conservative projection of evaluating those path expressions on the tree, i.e., every path in the projected tree may match at least one path expression in the set. The path expressions applied by TreeProject are

Table 3. Physical algebra.

Physical operators			Implementation
I/O	Parse(x:URI) : C (Tok)	=	parse(x)
	Serialize(x:URI,y:C (Tok)) : ()	=	serialize(x,unfold(y))
Conversion	Load(x:C (Tok)) : L (Tree)	=	load(x)
	Export(x:L (Tree)) : C (Tok)	=	export(x)
Construction			
	Sequence(x:C (Tok) , y:C (Tok)) : C (Tok)	=	concat(x,y)
	Element[q](x:C (Tok)) : C (Tok)	=	compose(fromList([$startElem(\mathtt{q},\mathtt{M}), hole, endElem$]), unmark(unfold(x)))
	Text(x:C (Tok)) : C (Tok)	=	fromList([$text(x)$])
Atomic	Scalar[a]() : a	=	Streaming agnostic
	Cast[a_0](x:a) : a_0	=	Streaming agnostic
Type	Validate[Type](x:C (Tok)) : C (Tok)	=	Streaming validation
	TypeMatches[Type](x:C (Tok)) : boolean	=	Streaming type matching
Navigation	TreeJoin[Step](x:C (Tok)) : C (Tok)	=	prune(markmap(nav$_{[Step]}$,x))
	TreeProject[pathpattern](C (Tok)) : C (Tok)	=	Streaming projection based on [22]
Functional	Var[q]() : *Xml*	=	Polymorphic over XML types
	Call[q](*Xml*, ...*Xml*$_n$) : *Xml*	=	Polymorphic over XML types
	Cond{*Xml*, *Xml*}(*boolean*) : *Xml*	=	Polymorphic over XML types
Tuple	MapFromItem{x:τ}(y:C (Tok)) : C (τ)	=	map(x,split(unfold(y)))
	CreateTuple, AccessTuple, ++, Select, Map, MapToItem, MapConcat, MapIndex, Join, GroupBy, OrderBy	=	Polymorphic over tuple field types C (Tok) and L (Tree) .

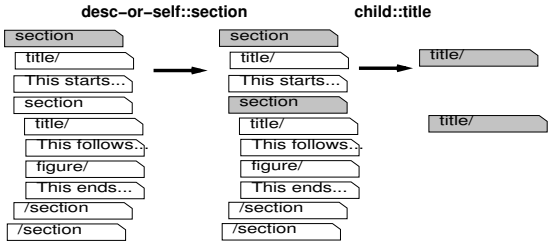

Figure 5. Applying steps to marked tokens

computed by a whole-plan path analysis described in previous work [22]. TreeProject is injected after Parse (See the plan in Figure 2), to reduce the size of the input to a plan. Since this operator was presented in previous work, we do not discuss it further here.

The TreeJoin implements the formal semantics of step expressions [11]. TreeJoin is a bulk navigation operator that takes a node sequence in document order (represented by a marked token cursor), and returns a node sequence in document order with no duplicates (also represented by a marked token cursor). A *strictly-forward* path expression can be implemented by the composition of TreeJoin operators, where a strictly-forward path only contains the *self*, *child*, *descendant*, *descendant-or-self*, and *attribute* axes with any node test but with no predicates. [3]

We first describe TreeJoin through an example, then give its definition in terms of micro-operators. Consider the expression:

[3]In practice, strictly-forward paths include predicates over attributes.

```
doc(URI)/descendant-or-self::section/child::title
```

which is compiled into the physical plan:

TreeJoin[*child::title*](
  TreeJoin[*descendant-or-self::section*](Parse(URI)))

Figure 5 depicts the result of applying the above plan to an input document. The Parse operator yields the document's root element, represented by a token cursor with the topmost token marked. The *descendant-or-self::section* step copies its input tokens to its output, erasing existing marks, and setting the mark on each $startElem(section)$ token. The *child::title* step simply copies all $startElem(title)$ tokens, their descendants, and corresponding $endElem$ token, observed at depth $d = 1$ relative to any marked token in its input and discards all other tokens. The resulting sequence of marked tokens is always in document order and contains no duplicates.

Note that some steps may produce token cursors with nested marks, e.g., `descendant-or-self`, denoting sub-trees that logically are copied to the output. Copying of nested sub-sequences is deferred as long as possible in a plan and depends on the semantics of subsequent operators. Most operators, including many built-in functions like `fn:count`, are defined on token cursors with nested marks. Three operators, Element, Serialize, and MapFromItem require that their inputs be unfolded.

In Table 3, TreeJoin is defined in terms of three micro-operators. The markmap operator logically applies its function argument f to each item in a sequence, which in

the physical data model, corresponds to applying f *independently* to each sub-sequence $s_1, \ldots, s_n$ delimited by a marked $startElem$ and the corresponding $endElem$. Because marked tokens may be nested, the sub-sequences $s_1, \ldots, s_n$ may overlap. The output of markmap is the *superposition* of all applications of f. In our framework, f is restricted to functions that copy all input tokens, possibly altering their marks. The results $f(s_1), \ldots, f(s_n)$ are combined as follows: Whenever a token t is marked in at least one $f(s_i)$, t is marked in the output. The markmap operator applies the micro-operator $\mathsf{nav}_{[step]}$ as described above; $\mathsf{nav}_{[step]}$ simply outputs the marked token cursor that results from applying the specified step to its input.

Lastly, the prune operator discards tokens that do not have any marked ancestors. Note that prune is *destructive*: It irretrievably discards tokens that are not contained within a matched tree node, thus matches are detached from their parents and siblings, as are the `title` elements in Figure 5. We choose to discard a match's context in favour of minimizing intermediate result sizes. However, this choice requires that subsequent operators in a plan do not depend on a node's context. Section 4 presents the analyses that enforce this constraint.

3.3 Tuple operators

All the tuple operators are polymorphic in their tuple field types, with the exception of MapFromItem, which converts a value in the tree fragment of the algebra to a value in the tuple fragment. MapFromItem takes an item sequence as input and yields one tuple for each item in the input. MapFromItem has two implementations: One for lists of trees and one for token cursors. The latter definition is in Table 3 and relies on the split and map micro-operators. The operator map is polymorphic and takes a function that maps an α value to a β value, a cursor of α values, applies the function to each α and returns a cursor of β values. In the definition of MapFromItem, the map takes a function, which constructs a tuple (x) and a cursor of *dependent* token cursors produced by the split operator. The split operator takes a token cursor $\mathbf{C}$ and splits it into distinguished sub-sequences of tokens, each sub-sequence corresponding to one tree node. It wraps each sub-sequence in its own token cursor $\mathbf{C}_i$, and returns a cursor that yields each of these token cursors in turn. Clearly, two such token cursors $\mathbf{C}_i$ and $\mathbf{C}_j$ are dependent because both draw tokens from $\mathbf{C}$.

Dependent cursors permit efficient pipelining of token-cursor values through tuple operators without requiring materialization, but they complicate the analysis that guarantees a plan is correct with respect to the construction and consumption of cursors. We address plan correctness and the analyses that guarantee it next.

4 Code Selection and Stream Analysis

Next, we describe how physical plans with streaming operators are selected and how to ensure the correctness of those plans. Given multiple physical representations of XML, the search space for selecting the physical operator for each logical operator in a plan becomes large. How to explore that search space and the development of corresponding cost models is future work. Our main focus here is on simple, yet efficient, code selection that ensures the correctness of physical plans.

4.1 Code selection

In this section, Op denotes a plan in the logical algebra from [26], and POp is a physical plan in the physical algebra described in Section 3. Code selection is a mapping from a logical plan to a physical plan: $\mathtt{CS}(\mathsf{Op}) \rightarrow \mathsf{POp}$. For a given logical plan, CS is defined on every sub-plan, i.e., it defines a mapping for every logical operator.

We denote by $[\![\mathsf{Op}]\!]$ the result of evaluating the logical operator Op in the logical data model. $[\![\mathsf{POp}]\!]_p$ is defined similarly on the physical data model. Let Δ be a mapping from physical values to logical values as in Section 2, and let $\cong$ denote deep-equality over XML trees. Given these functions, correctness of physical plans is defined as:

Definition 4.1: $\mathtt{CS}(\mathsf{Op}) = \mathsf{POp}$ is a *correct* physical plan for Op iff for each Op_i a subplan of Op:

$$\Delta([\![\mathtt{CS}(\mathsf{Op}_i)]\!]_p) \cong \Delta([\![\mathsf{Load}(\mathtt{CS}(\mathsf{Op}_i))]\!]_p) \cong [\![\mathsf{Op}_i]\!]$$

■

Intuitively, a physical plan is correct if each of its subplans yields the same logical value as the value produced by the sub-plan followed by materialization. We use deep equality to compare values, because the result of a streaming plan yields a tree without its parental or sibling context.

We now define the *stream-safety* property, which is sufficient to ensure the correctness of a physical plan and can be inferred through static analysis.

Definition 4.2:[Stream Safety] A logical (sub-)plan Op is stream safe with respect to a whole plan Op_0 iff it satisfies the following conditions:

1. In Op_0, navigational access on the XML values returned by Op is strictly forward.
2. In Op_0, the tuples returned by Op are consumed in the *same order* in which they were created.
3. In Op_0, the fields of tuples returned by Op are accessed *at most* once.

■

The first condition is checked using an existing path analysis [22], which computes an approximation of all paths

that access data in a query. The second condition is always true under the assumption that all algebraic operators that reorder tuples materialize the contents of their tuple fields. This constraint seems strong, but most pipelining operators process tuples in input order. This constraint need only be enforced on the blocking operators: OrderBy, GroupBy, and the right-hand side of hash and sort joins. To check the third condition, Section 4.2 presents a data-flow analysis that for each tuple field, computes a worst-case estimate of the number of times it is accessed during plan evaluation.

Our code-selection heuristics are based on the following assumptions: (1) Conversion between physical representations is expensive; (2) When accessing streamed sources, streaming operators are more efficient than materialization followed by operators on the materialized representation; (3) Copying whole subtrees is expensive and should be avoided. Based on these assumptions, code selection applies the following rules to each sub-plan Op of a whole plan Op_0, bottom-up:

1. If (a) the selected physical operators for the input(s) of Op are streamed, (b) a streaming operator POp exists for Op, and (c) Op is stream safe in Op_0, then `CS(`Op`)` selects POp. Otherwise, `CS(`Op`)` uses a materialized operator.

2. If Op is a constructor operator, `CS(`Op`)` uses a streaming operator.

We always use the streaming variant of constructors because copying trees is strictly more expensive than producing a lazy stream. Recall from Figure 2 that streaming operators are applied to the source `$bids` up to the point where data from this source is bound to a tuple field (`uid`) that is accessed more than once in the rest of the plan, and streaming operators are used in the part of the plan that constructs the result and serializes it.

Theorem 1 (Code Selection Correctness) *Physical plans generated through the above code selection algorithm are correct streaming plans.*

Due to limited space, we give the intuition for the proof. The key part of the proof is to show that stream safety is sufficient to ensure correctness. The proof proceeds by induction over the operators in an algebraic plan. The first part of the proof relates stream safety to correctness. Stream-safety condition 1 ensures that a node's parent and sibling context is discarded only if that context is not required by later operators in the plan. Stream-safety condition 2 forces dependent cursors to be consumed without violating the dependencies among them, and is always true, because they are only fed to pipelined operators. Lastly, stream-safety condition 3 is a consequence of the fact that cursors are mutable. The second part of the proof checks that the use-count analysis in Section 4.2 correctly infers an upper bound of the actual usage count, ensuring that condition 3 is always satisfied through code selection.

4.2 Use Count Analysis

We define a data-flow analysis [1] that computes the tuple-field use counts of a plan by combining the use-counts computed for sub-plans. The main difference from standard data-flow analysis is that the analysis must account for the implementation semantics of each physical operator. In particular, simply counting all occurrences of operators that access tuple fields (e.g., AccessTuple[q]) is not sufficient, because implementations of some operators make copies of input tuples. For example, the MapConcat operator, which is a dependent product, makes (virtual) copies of tuples from its independent input. As a result, subsequent access to the tuple fields processed in a MapConcat must be counted multiple times. Our analysis, therefore, tracks the provenance of tuple fields.

The analysis is specified using inference rules. Let Q be the set of all tuple field names; $RF \subseteq Q$; $CF \subseteq Q$; an environment $Env = (Env_{CF} \subseteq Q, Env_{RF} \subseteq Q)$; and $UF \subseteq (Q \times \{0, 1, \infty\})$. The following judgment holds iff operator Op uses the fields UF and returns the fields RF under the environment Env:

$$Env \vdash \mathsf{Op} \Rightarrow (UF, CF, C) \ \texttt{returns}\ RF$$

The environment keeps track of tuple fields' usage when tuples are passed to dependent operators in a plan. $C \in \{1, \infty, NoTable\}$ is a conservative estimate of the number of tuples produced by Op. Tuple operators produce 1 or ∞, and tree operators produce $NoTable$. CF is a set of candidate fields for which any subsequent access means an effective iterated access.

Table 4 contains the inference rules for selected tuple operators with a focus on operators that have implicit iteration. The three rules in the first column are straightforward. The first rule returns the usage count for the input tuple (IN) from the environment. In the second rule, creating a tuple with one field q returns q and passes on the use counts of its input Op. In the third rule, if field q is already a candidate field, we count multiple accesses, otherwise just one.

The second column contains the rules for Map and MapConcat. The Map operator is implemented by the polymorphic map operator applied to tuple cursors. For each tuple returned by Op1, it binds IN to the given tuple, and evaluates its dependent branch Op2. Thus, the environment Env' for inferring Op2's usage counts depends on the candidate and return fields obtained by analyzing Op1. The use counts for Map depend on both Op1 and Op2, and their fields are merged to produce the analysis result. The following table defines $\uplus$ for merging individual use counts. We informally extend $\uplus$ over sets of use counts, merging pairs with matching field names.

Table 4. Use-count inference rules for selected tuple operators

$$\frac{Env = (CF, RF)}{Env \vdash \mathsf{IN} \Rightarrow (\emptyset, CF, 1)\ \texttt{returns}\ RF}$$

$$\frac{Env \vdash \mathsf{Op} \Rightarrow (UF, CF, C)\ \texttt{returns}\ RF}{Env \vdash \mathsf{CreateTuple}[q](\mathsf{Op}) \Rightarrow (UF, CF, 1)\ \texttt{returns}\ \{q\}}$$

$$\frac{Env = (CF, RF) \quad UF = \texttt{if}\ (q \in CF)\ \texttt{then}\ \{(q, \infty)\}\ \texttt{else}\ \{(q, 1)\}}{Env \vdash \mathsf{AccessTuple}[q] \Rightarrow (UF, \emptyset, NoTable)\ \texttt{returns}\ \emptyset}$$

$$\frac{Env \vdash \mathsf{Op1} \Rightarrow (UF_1, CF_1, C_1)\ \texttt{returns}\ RF_1 \quad Env' = (CF_1, RF_1) \quad Env' \vdash \mathsf{Op2} \Rightarrow (UF_2, CF_2, C_2)\ \texttt{returns}\ RF_2}{Env \vdash \mathsf{Map}\{\mathsf{Op2}\}(\mathsf{Op1}) \Rightarrow (UF_1 \uplus UF_2, CF_1 \cup CF_2, max(C_1, C_2))\ \texttt{returns}\ RF_2}$$

$$\frac{Env \vdash \mathsf{Op1} \Rightarrow (UF_1, CF_1, C_1)\ \texttt{returns}\ RF_1 \quad Env' = (CF_1, RF_1) \quad Env' \vdash \mathsf{Op2} \Rightarrow (UF_2, CF_2, C_2)\ \texttt{returns}\ RF_2 \quad \mathbf{CF_3} = \texttt{if}\ (\mathbf{C_2 > 1})\ \texttt{then}\ \mathbf{RF_1}\ \texttt{else}\ \emptyset}{Env \vdash \mathsf{MapConcat}\{\mathsf{Op2}\}(\mathsf{Op1}) \Rightarrow (UF_1 \uplus UF_2, CF_1 \cup CF_2 \cup \mathbf{CF_3}, max(C_1, C_2))\ \texttt{returns}\ RF_1 \cup RF_2}$$

$$\frac{\begin{array}{c} Env \vdash \mathsf{Op1} \Rightarrow (UF_1, CF_1, C_1)\ \texttt{returns}\ RF_1 \quad Env \vdash \mathsf{Op2} \Rightarrow (UF_2, CF_2, C_2)\ \texttt{returns}\ RF_2 \\ CF_1' = \texttt{if}\ (C_2 > 1)\ \texttt{then}\ RF_1\ \texttt{else}\ \emptyset \quad CF_2' = \texttt{if}\ (C_1 > 1)\ \texttt{then}\ RF_2\ \texttt{else}\ \emptyset \\ \mathbf{CF} = \mathbf{CF_1} \cup \mathbf{CF_2} \cup \mathbf{CF_1'} \cup \mathbf{CF_2'} \quad Env' = (CF, RF_1 \cup RF_2) \\ Env' \vdash \mathsf{Op3} \Rightarrow (UF_3, CF_3, C_3)\ \texttt{returns}\ RF_3 \\ UF = \{(rf, 1) \mid rf \in RF_2\} \uplus UF_1 \uplus UF_2 \uplus UF_3 \end{array}}{Env \vdash \mathsf{Join}\{\mathsf{Op3}\}(\mathsf{Op1},\ \mathsf{Op2}) \Rightarrow (UF, \mathbf{CF}, max(C_1, C_2))\ \texttt{returns}\ RF_1 \cup RF_2}$$

x	y	$x \uplus y$	x	y	$x \uplus y$
$(q,0)$	$(q,0)$	$(q,0)$	$(q,1)$	$(q,1)$	(q,∞)
$(q,0)$	$(q,1)$	$(q,1)$	(q,∞)	(q,u)	(q,∞)
$(q,1)$	$(q,0)$	$(q,1)$	(q,u)	(q,∞)	(q,∞)

The rule for MapConcat is identical to that of Map except for the highlighted (last) premise. MapConcat computes a dependent product of the tuples produced by its branches. Logically, the tuples in the independent branch Op1 are copied as many times as there are tuples returned in the dependent branch Op2. To account for this potential iteration, we propagate Op1's returned fields as *candidate fields* to subsequent operators only if the cardinality estimate of C_2 is > 1. If the cardinality estimate of C_2 is ≤ 1, then each of Op1's tuples is accessed at most once.

The rule in the third column defines use counts for a *nested-loop* join. We propagate the returned fields of the two independent branches Op1 and Op2 (RF_1 and RF_2) as candidates fields to the dependent branch Op3 for the same reason as we did in MapConcat. The last premise in this rule reveals a tricky detail. A nested-loop join materializes the right-hand side input (Op2) in order to avoid repeated evaluation, so we have to count a single access to each field in the right-hand side tuple, because materializing a token cursor to a tree consumes that cursor.

The rule for hash join is identical to that of nested-loop join except for the highlighted (fifth) premise. In the rule for hash join, we do not need to propagate CF_1' or CF_2' as candidates to the dependent branch, because all accesses inside the join predicate Op3 are guaranteed to be evaluated only once, either when constructing the hash table from the right-hand branch or when probing the hash table from the left-hand branch. The candidate fields CF_1' and CF_2' are propagated to CF in the conclusion, as they are in the rule for nested-loop join.

5 Experimental Evaluation

Next, we assess the impact of streaming operators on plans that can be partially or completely streamed. First, we verify that for queries that can be completely streamed, performance scales linearly with both query size and data size. Second, we evaluate the impact of streaming on the XMark benchmarks and our motivating query in Figure 1. All experiments were run on one machine: an Intel(R) Pentium(R) 4 CPU 2.00GHz, with 0.5GB RAM, running Linux version 2.6.12. The physical algebra and algorithms are implemented in Galax development version 0.6.6.

We ran the sets of queries in Figure 6 on documents of increasing size to show that the token-cursor operators scale linearly with both query and data size. We used the XCheck [15] performance-evaluation platform, which runs each experiment four times and averages the last three (hot) runs. The reported times include document loading, query compilation and query execution but exclude document serialization, which may not scale linearly when unfold is applied to nested marked tokens. Although query compilation does not scale linearly, its absolute overhead was trivial. We report loading time, because we assume that in applications that consume streamed XML input, the opportunity to preload a document will not exist, therefore loading time cannot be amortized over multiple queries.

The $\mathsf{Q1}_k$ set of queries verifies the scalability of simple path expressions. For strictly-forward path expressions, no unfolding occurs, so we expect linear scalability for both data and query sizes. We use five MemBeR [2] documents containing 1 M to 5 M nodes, corresponding to document sizes of 22 MB to 114 MB. Figure 8 plots the execution

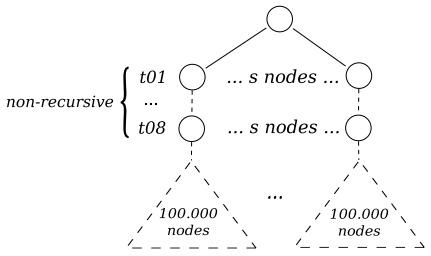

Figure 7. MemBeR documents

Q1$_k$ doc()/descendant-or-self::node()
with step repeated k times, for $1 \leq k \leq 8$

Q2$_k$ for \$x in P_k return \$x/descendant::node()
where P_k is **Q1**$_k$, for $1 \leq k \leq 8$

Q3$_1$ for \$x in \$input/descendant::t01 return \$x/descendant::t02

Q3$_2$ for \$y in (for \$x in \$input/descendant::t01
return \$x/descendant::t02)
return \$y/descendant::t03

Q3$_k$ Increase nesting in **Q3**$_2$ for $3 \leq k \leq 8$

Q4$_1$ element a {doc()/descendant-or-self::node()}

Q4$_2$ element a { element b { doc()/descendant-or-self::node()/descendant-or-self::node() }}

Q4$_k$ Increase number of constructors plus steps in **Q4**$_2$ for $3 \leq k \leq 8$

Figure 6. Queries to verify scalability

time for increasing query sizes and input sizes for **Q1**$_k$. As expected, execution time scales linearly in both parameters.

Due to space constraints, we do not present graphs for the following experiments, but we report that they scale linearly in both query and document sizes, as expected. The **Q2**$_K$ set of queries yields plans with tuple Map operators interleaved with TreeJoin. These queries verify that streaming tuple Map's have no impact on the combined scalability of nested operators.

When Map's are used to retrieve descendant nodes and when the input nodes in turn have ancestor-descendant relationships among them, unfolding is required for proper evaluation. Unfolding, however, is a buffering and potentially quadratic-time operator that can jeopardize the value of a streaming map operator. However, when the queried fragment of the input document does not contain recursive elements, unfolding is the identity function, and linear query and data scalability should be preserved. We verify this property by running **Q3**$_k$ over documents from 25 MB to 125 MB for which the selected part contains no recursive elements. Figure 7 depicts the shape of these documents. The number of siblings s varies from 10 to 50. The leaf trees contain 100,000 nodes with recursively nested tags, different from the tags $t01 \ldots t08$.

The query set **Q4**$_k$ verifies that composed constructors scale linearly in output size. Constructors do not necessarily scale linearly with input size, because they unfold their contents. If a constructor's contents does not contain nested marked tokens, then they indeed scale linearly with input size. We verified this property on five MemBeR generated documents of depth 7, with 10 tags uniformly distributed over the tree, varying in size from 14MB to 22MB.

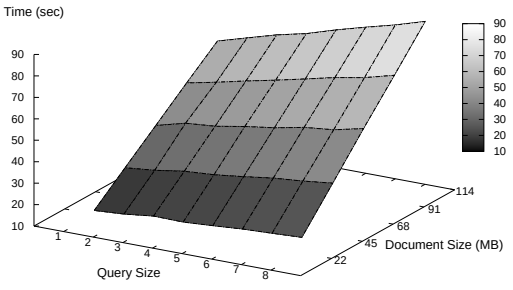

Figure 8. Results for query set **Q1**$_k$

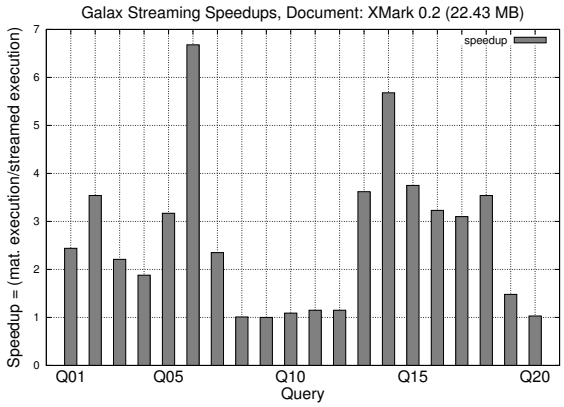

Figure 9. Speedup of XMark execution times

We also ran the XMark benchmark suite, comparing streaming plans to fully materialized plans. The results of running the queries, excluding serialization time are shown in Table 5 and Figure 9. All queries, with the exception of Queries 8 to 12 and Query 20, show a substantial improvement. The queries that are fully streamable (2, 6 and 15) typically have significant speedups, and eliminating unnecessary materialization in fragments of queries also yields measurable improvements (1, 4, 5, 7, 14 and 16–19).

The XMark queries that express joins (Queries 8–12) do not benefit much from streaming, due in part to their self-join semantics, which requires materialization of large parts of the input document, and also due to Galax's inability to select the best join plan for these specific queries. A similar problem arises in Query 20 in which a function call limited the use of a streaming evaluation approach.

To demonstrate the potential of the streaming approach on complex queries, we ran the query Q1 in Figure 1, which joins two separate XMark-based files, and allows one of the inputs (`bids.xml`) to be streamed. The join is computed using a hash-join algorithm, where `persons.xml` was 11 MB in size and `bids.xml` ranged from 6.5 MB to

Table 5. Absolute execution times (secs) for XMark streaming plans on a 22 MB document

Q1	9.44	Q6	6.07	Q11	1532.90	Q16	6.25
Q2	5.85	Q7	26.06	Q12	1536.93	Q17	6.78
Q3	9.62	Q8	21.29	Q13	5.67	Q18	5.79
Q4	14.4	Q9	69.06	Q14	8.59	Q19	30.71
Q5	6.36	Q10	31.91	Q15	5.46	Q20	22.74

100 MB. We observed that the streaming approach scales much better with the input size and has a much smaller memory footprint. As a result, the streaming plan handled inputs greater than 100 MB, whereas the materialized plan failed for input sizes greater than 20 MB.

6 Conclusion

We presented a physical algebra for XQuery that allows the generation of evaluation plans that blend streaming techniques with evaluation and optimization techniques over indexed XML documents. The originality of our approach is its ability to use XML streaming evaluation by almost directly relying on traditional relational pipelining, offering important benefits for a reduced development cost. We believe this work is an important first step in bridging the gap between streaming evaluation [3, 9, 13, 19, 20] and more traditional evaluation techniques [4, 23, 25, 26] for XML. In the future, we are interested in extending our approach to cover more advanced streaming techniques [9, 12, 19, 16], which notably include stream buffers and the ability to consume the same stream multiple times.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Priciples, Techniques and Tools*. Addison Wesley, 1986.

[2] L. Afanasiev, I. Manolescu, and P. Michiels. MemBer: A micro-benchmark repository for XQuery. In *XSym 2005*, v. 3671 of *LNCS*, pp 144–161. Springer, 2005.

[3] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *ICDE*, pp 455–466, 2003.

[4] K. Beyer, R. J. Cochrane, V. Josifovski, et al. System RX: one part relational, one part XML. In *SIGMOD*, pp 347–358, 2005.

[5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. Candidate Recommendation, June 2006.

[6] S. Bose, L. Fegaras, D. Levine, and V. Chaluvadi. A query algebra for fragmented XML stream data. In *DBPL*, pp 195–215, 2003.

[7] M. Branter, S. Elmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. In *ICDE*, 2005.

[8] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, pp 310–321, 2002.

[9] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML stream query processor SPEX. In *ICDE*, pp 1120–1121, 2005.

[10] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, pp 168–179, 2004.

[11] D. Draper, P. Fankhauser, M. Fernandez, et al. XQuery 1.0 and XPath 2.0 formal semantics, W3C working draft. Candidate Recommendation, June 2006.

[12] L. Fegaras, R. Dash, and Y. Wang. A fully pipelined XQuery processor. XQuery Implementation, Experience and Perspectives (XIME-P) Workshop, 2006.

[13] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *VLDB*, pp 997–1008, 2003.

[14] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *CIKM*, pp 784–791, 2005.

[15] M. Franceschet, E. Zimuel, L. Afanasiev, and M. Marx. XCheck, a platform for benchmarking XQuery processors, 2006. http://ilps.science.uva.nl/Resources/XCheck.

[16] A. Frisch and K. Nakano. Streaming XML transformations using term rewritings. Draft Manuscript, July, 2006.

[17] T. Grust and J. Teubner. Relational algebra: Mother tongue — XQuery: Fluent. In *Proc of the 1st Data Management Workshop on XML Databases*, 2004.

[18] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its axis steps. In *VLDB*, pp 524–535, Berlin, Germany, Sept. 2003.

[19] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD*, pp 419–430, 2003.

[20] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *VLDB*, pp 228–239, 2004.

[21] I. Manolescu and Y. Papakonstantinou. XQuery midflight: Emerging database-oriented paradigms and a classification of research advances. In *ICDE*, page 1143, 2005.

[22] A. Marian and J. Simeon. Projecting XML documents. In *VLDB*, pp 213–224, 2003.

[23] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *ICDE*, pp 239–250, 2004.

[24] P. Michiels, G. Mihaila, and J. Siméon. Put a Tree Pattern in your Tuple Algebra In *ICDE*, to appear, 2007.

[25] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, pp 71–82, 2004.

[26] C. Re, J. Simeon, and M. Fernandez. A complete and efficient algebraic compiler for XQuery. In *ICDE*, 2006.

[27] M. Wei, M. Li, E. Rundensteiner, M. Mani. Processing Recursive XQuery over XML Streams: The Raindrop Approach. In *XSDM*, pp 89–98, 2006.